



PROYECTO DE SISTEMAS INFORMÁTICOS

CURSO 2010/2011

Desarrollo de agentes inteligentes para videojuegos en primera persona

Autores:

Duchini Ordoñana, Alejandro

Herrero Herrera, Eric

Director:

González Calero, Pedro Antonio

Agradecimientos

En primer lugar queremos agradecer este proyecto a nuestro profesor por haber confiado en nosotros para llevarlo a cabo y habernos indicado el camino a seguir en los momentos en los que las cosas se ponían cuesta arriba y parecían no tener salida.

A nuestras familias y amigos de siempre por haber aguantado estoicamente nuestros horarios a lo largo de toda la carrera y por habernos apoyado en todo momento. Sin este apoyo es posible que no hubiéramos llegado hasta este punto.

A nuestros compañeros y amigos de la facultad por haber estado ahí en las situaciones peliagudas y en los momentos de relax y por haber compartido con nosotros sufrimientos y alegrías.

A nuestros profesores por enseñarnos que las recompensas no se ganan sin esfuerzo ni dedicación y por darnos una visión de futuro sobre lo que nuestra carrera puede brindarnos el día de mañana.

Y por último a nosotros mismos, por haber trabajado tan duramente por sacar adelante estos años de carrera y por quedarnos con un buen sabor de boca por un trabajo bien hecho.

Índice

1. Resumen	08
2. Abstract	09
3. Palabras Clave	11
4. Formas de programar IAs	13
4.1. Máquinas de Estados	13
4.2. Árboles de Comportamiento	16
4.2.1. Java Behaviour Trees	18
4.2.2. Pasos para construir en JBT un BT	19
4.2.3. Ejemplo de un modelo de BT	19
5. Unreal Tournament 2004	23
6. Pogamut	26
6.1. Como conectar el Pogamut con el UT2004	26
7. Un Bot en Pogamut	31
7.1. Módulos básicos de un Bot	31
7.2. Estructuras básicas de un Bot	33
7.2.1. Método “prepareBot”	34
7.2.2. Método “getInitializeCommand”	34
7.2.3. Método “botInitialized”	34
7.2.4. Método “botSpawned”	35
7.2.5. Método “logic”	35
7.2.6. Método “botKilled”	35
8. Comunicación entre Bots	37
8.1. Buzones	37

Índice

9. Nuestro Bot	40
9.1. La estrategia	40
9.2. El árbol de decisiones	41
9.2.1. Árbol Madre	42
9.2.2. Árbol Capturar	43
9.2.3. Árbol Proteger	44
9.2.4. Árbol Defender	45
9.2.5. Árbol Recuperar	46
9.3. El árbol de acciones	47
9.3.1. Atacar	48
9.3.2. Dejar de Disparar	49
9.3.3. Rotar	49
9.3.4. Perseguir	49
9.3.5. Coger salud	49
9.3.6. Coger ítem	50
9.4. Los métodos básicos del Bot	50
9.4.1. Método "prepareBot"	50
9.4.2. Método "getInitializeCommand"	50
9.4.3. Método "botInitialized"	51
9.4.4. Método "botSpawned"	51
9.4.5. Método "logic"	51
10. Conclusiones	53
10.1. Árboles de comportamiento VS máquinas de estado	53
Glosario	55
Bibliografía y Referencias	57

1. Resumen

Este proyecto ha sido orientado al campo de la Inteligencia Artificial aplicada en el mundo de los videojuegos. De forma más específica, ha consistido en la investigación de diversas técnicas aplicadas al desarrollo de agentes inteligentes para videojuegos en primera persona, realizando comparaciones entre la ya explotada técnica de las máquinas de estado y la reciente adaptación de los árboles de comportamiento en este campo. Estas comparaciones abarcan el diseño, la implementación y el depurado de ambas técnicas de modelado de la Inteligencia Artificial.

Para poder llevar a cabo este proyecto, nos apoyamos principalmente en el juego Unreal Tournament 2004 y en la herramienta Pogamut, la cual nos permite y facilita el desarrollo de la Inteligencia Artificial de los bots del Unreal Tournament sin tener que preocuparnos de otros aspectos. Otra herramienta que nos es de vital importancia para poder diseñar y desarrollar los árboles de comportamiento que integraremos en el Pogamut es la denominada Java Behaviour Trees. Tanto ésta como Pogamut son dos herramientas que están implementadas bajo Java.

En un principio se diseñó y desarrolló un modelo de Inteligencia Artificial para distintos bots basado en una máquina de estados. Esto nos sirvió para podernos adaptar a la herramienta Pogamut y observar el comportamiento de los agentes en una partida de tipo Death Match del Unreal Tournament. Más tarde suplimos esta técnica de modelado por un árbol de comportamiento, poniendo los aspectos antes mencionados (diseño, desarrollo y depuración) en contraposición a los de la máquina de estados.

Una vez vista la potencia que puede llegar a tener un árbol de comportamiento, nos embarcamos con esta técnica en el modelado de una estrategia que haga posible la cooperación de varios agentes de un mismo equipo para poder conseguir desenvolverse en una partida de Captura de Bandera.

2. Abstract

Our Project has been oriented to the Artificial Intelligence field applied to the gaming world. Particularly, it is based on the research of various techniques applied to the development of intelligent agents for first person videogames, comparing between the already greatly used Finite State Machine technique and the recent adaptation of behavioral trees to this field. These comparisons include the design, implementation and debugging of both techniques.

To accomplish this goal, we mainly based on the game Unreal Tournament 2004 and on the Pogamut tool, which makes the development of bots' AI in the game easier for us not minding other aspects. Another very important tool we use to design and develop behavioral trees which we'll integrate into Pogamut is the so-called Java Behaviour Trees. Both of them are programmed in Java.

At first, we designed a Finite State Machine based AI model for some bots. This served us to get used to the Pogamut tool and to observe the agent's behaviour in a Death Match game type. Later on, we substituted this modeling technique to a behaviour tree, comparing the previously mentioned aspects (design, implementation and debugging) with the Finite State Machine ones.

Once we saw how powerful behavioural tree could be, we used this technique to model a strategy that enables the cooperation of various agents of the same team in a Capture the Flag game type.

3. Palabras Clave

- Inteligencia Artificial (IA)
- Agente (BOT)
- Máquina de estados (FSM)
- Árboles de comportamiento (BT)
- Unreal Tournament 2004 (UT2004)
- Pogamut
- Java Behaviour Trees (JBT)

4. Formas de programar IAs

En un inicio, cuando empezaron a aparecer los primeros juegos y antes de las generalizaciones del uso de técnicas de IA, los personajes que no eran controlados por un ser humano sino por la propia máquina (NPCs) se comportaban de una manera fija, realizando desplazamientos según una ruta predeterminada o al azar, disparando de forma aleatoria, incluso siendo incapaces de evaluar situaciones de peligro. Estos comportamientos carentes de IA alguna hacían que el juego resultase monótono y predecible llegando a aburrir al jugador dado que éste no se sentía retado por el propio juego.

El intentar resolver dichos problemas para poder hacer los juegos más atractivos de cara al jugador hace que se empiecen a investigar y evolucionar las técnicas existentes para poder programar una IA para los NPCs que sea más competitiva. Esta IA se puede implementar de diversas formas, dependiendo el tipo de técnica que se siga: Máquinas de Estado (FSM, Finite-State Machines), Búsqueda de Caminos, Heurísticas, Algoritmos Genéticos, Redes Neuronales, Lógica Difusa.

Una de las ramas principales de este proyecto es el estudio y comparación entre dos técnicas que nos permitan realizar una IA de forma competente destinada a los bots (NPCs) del videojuego en primera persona Unreal Tournament 2004. Estas dos técnicas son, las Máquinas de Estados por un lado, dado que típicamente la IA de un bot suele implementarse mediante esta técnica (dentro del ámbito de desarrollo de videojuegos), en contraposición con los Árboles de Comportamiento, técnica derivada de los FSM, con el fin de simplificar la implementación de éstos gracias a la abstracción de conceptos.

4.1. Máquinas de Estados

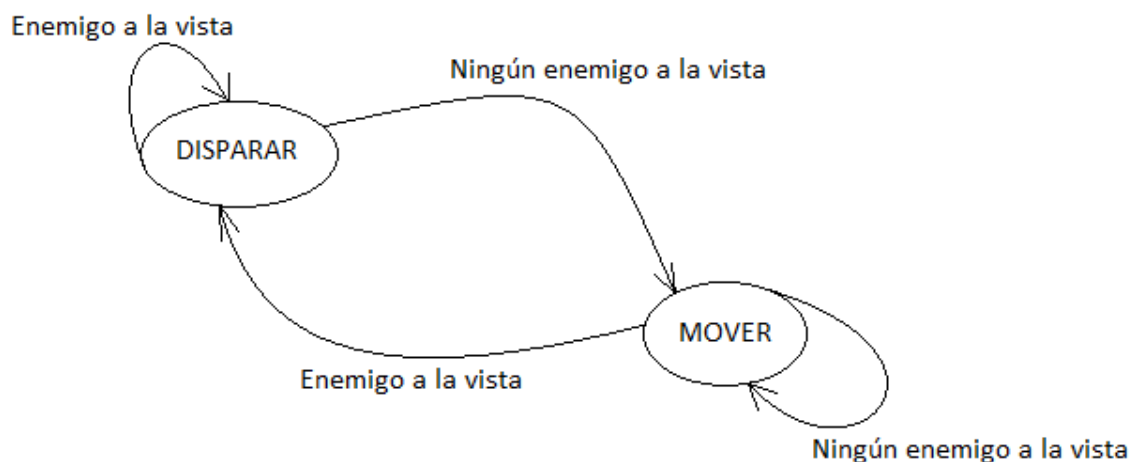
Una máquina finita de estados (en inglés FSM) es una entidad abstracta compuesta por un conjunto de estados (vértices) y de transiciones (aristas) entre dichos estados. Las transiciones se producen según unas condiciones que dependen del exterior y opcionalmente del estado actual. A su vez, la máquina genera una serie de acciones según el estado actual en el que se encuentre.

Por lo general, las FSM se usan para manejar unidades a nivel individual. Teniendo en cuenta lo que queremos que hagan dichas unidades y de qué manera lo realicen, agrupando acciones por un lado y condiciones de activación por otro.

En nuestro caso, la idea es que el bot se vaya moviendo de estado en estado, teniendo en cada estado una acción que realizar del tipo: moverse, atacar, perseguir, curarse, etc. Estas acciones que realiza son excluyentes, no interfieren unas con otras, dado que cada estado es independiente de los demás. El movimiento entre estos estados se da gracias a las transiciones. Estas transiciones se cumplen cuando se activan ciertas condiciones de activación como son: ningún enemigo a la vista, enemigo a la vista, enemigo perdido, salud baja, etc.

El problema que plantea una máquina de estados son las transiciones entre ellos. Para cada cambio que pueda percibir el bot deberá hacerse una transición de estado, ya sea a otro distinto o al mismo en el que se encontraba. Esto implica que para máquinas de estados grandes, se produce una explosión combinatoria debido a que para cada estado tiene que haber una transición distinta por cada uno de los cambios posibles en el entorno del bot, lo que ocasiona un cómputo demasiado grande, ya que cada vez que se ejecuta un estado, se deben comprobar todas y cada una de las condiciones relacionadas con cada transición para decidir si debe haber un cambio de estado o no, lo que puede retrasar bastante este cambio de estado.

A continuación exponemos un ejemplo de una máquina de estados sencilla que podría ser implementada por nuestro bot:

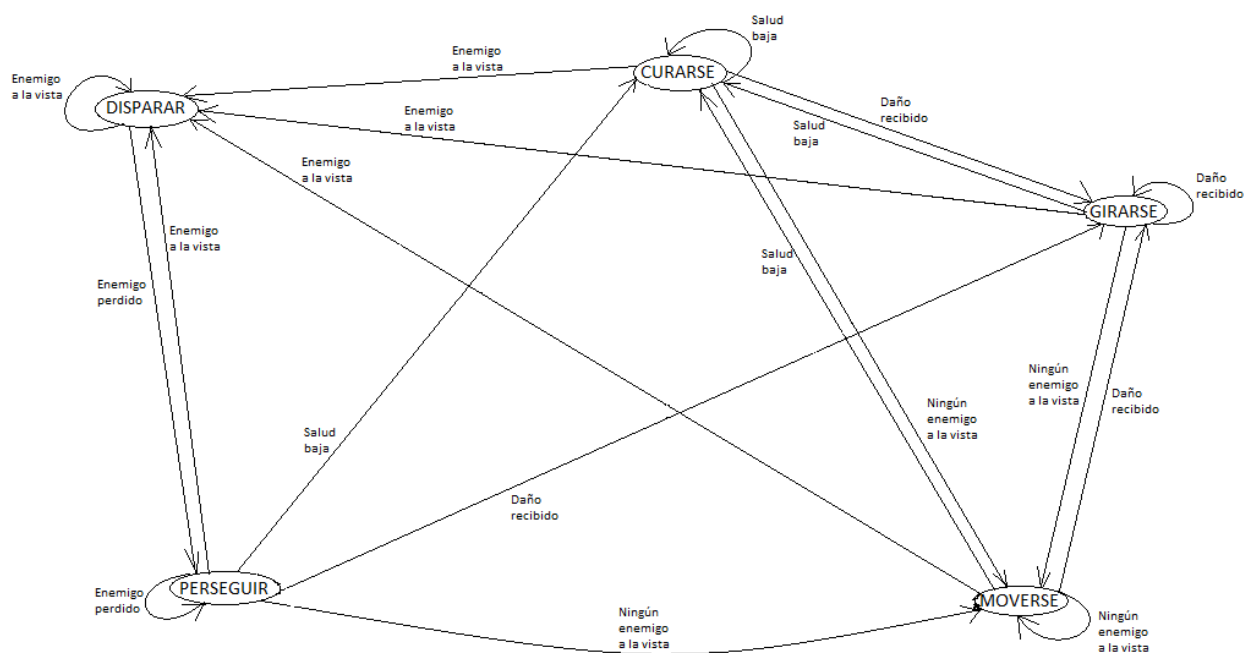


Ejemplo 1: Máquina de Estados sencilla.

Como se puede observar, es una máquina de estados muy sencilla en la que simplemente hay un par de estados y una transición desde cada estado a otro en

función de si el bot ve a un enemigo o no (una transición distinta por cada estado existente en la máquina). Se puede también apreciar que es posible que haya transiciones de un estado a sí mismo.

El ejemplo anterior es un ejemplo muy sencillo, con sólo dos estados y un par de transiciones, pero si queremos refinar el comportamiento del bot y hacerlo un poco más eficaz, debemos complicar la máquina de estados añadiéndole algunos más (ver Ejemplo 2).



Ejemplo 2: Máquina de Estados compleja.

Como se puede ver, a pesar de no haber transiciones de todos los estados a todos, el cálculo del cambio de estado ya se complica y para estados como “Perseguir” o “Moverse”, puede resultar costoso y largo.

Resumiendo, una máquina de estados tiene la ventaja de poder descomponer una acción compleja en sub-acciones de carácter más simples. Pudiendo diseñar y estructurar complejos comportamientos de forma sencilla. Esta estructuración hace que se encapsule el comportamiento facilitando su depuración a la hora de encontrar algún fallo, siendo muy eficaz el empleo de esta técnica cuando queremos construir máquinas pequeñas y simples (alrededor de 10-15 estados por máquina). La gran desventaja que tiene una máquina de estados es cuando empiezan a incrementarse el número de estados y transiciones a causa de intentar construir bots más complejos. El mantenimiento de esta máquina comienza a ser insostenible, como ya hemos visto en el ejemplo anterior. Este inconveniente de las máquinas de estados se intenta

solucionar mediante los Árboles de Comportamiento. Los cuales siguen manteniendo las ventajas que ofrece los FSM pero a su vez permiten una fácil implementación de IAs complejas.

4.2. Árboles de Comportamiento

Los árboles de comportamiento (BTs) se han convertido en una herramienta muy popular para programar la IA de los NPCs. Halo 2 fue el primer videojuego de alto standing que implementó y detalló el uso de BTs para su IA. Esto hizo que se expandiera esta técnica en el ámbito de los videojuegos ya que después de que el Halo 2 tuviera tan buen resultado, muchos otros comenzaron a utilizarla para el desarrollo de sus IAs.

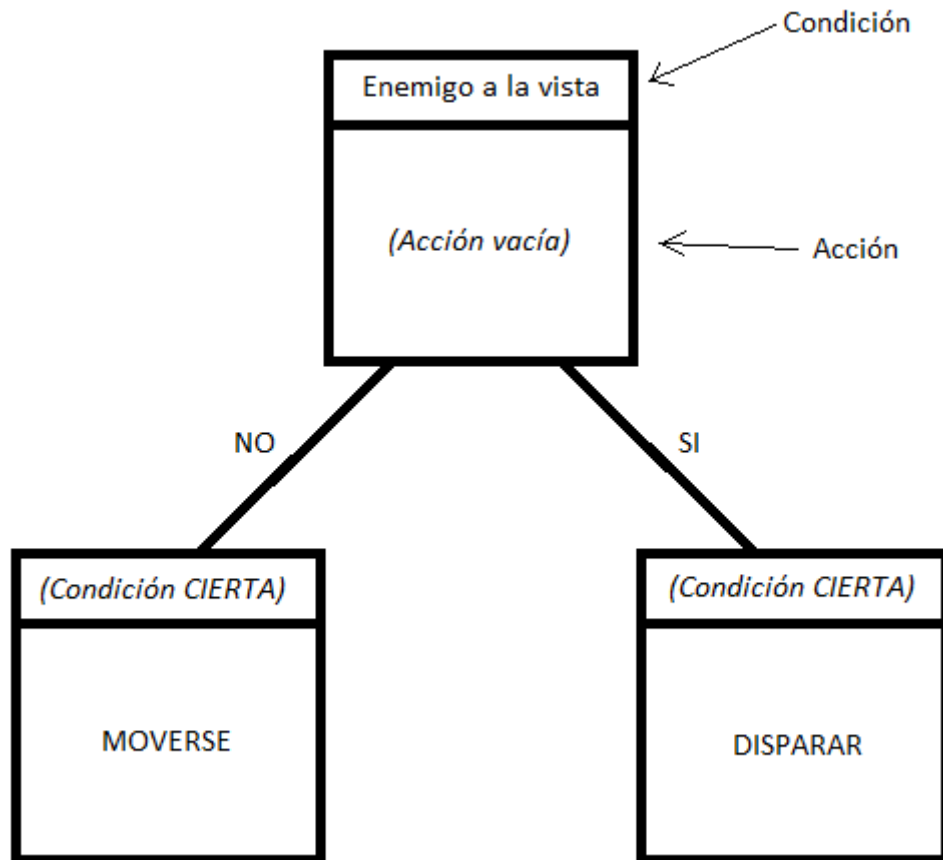
Esta buena acogida que están teniendo los BTs es gracias a la facilidad que brindan para plasmar y estructurar los conceptos claves que se quiera que adopten los NPCs, siendo posible la realización/creación del diseño de estas estructuras tanto por programadores como no programadores. Los BTs son una síntesis de unas cuantas técnicas de que son utilizadas para el desarrollo de IAs: máquinas de estado, organización, planificación y acción ejecución.

Los BTs tienen mucho en común con las FSM, como hemos podido apreciar en el apartando anterior, aunque en lugar de estados, los BTs tienen como bloque principal una tarea. Esta tarea puede ser tan sencilla como comprobar el valor de una variable en un estado del juego, o ejecutar una animación.

Las tareas pueden estar compuestas por sub-árboles para poder representar acciones más complejas. A su vez, estas acciones complejas pueden ser utilizadas para componer un comportamiento que tenga una mayor abstracción. En esta composición de distintos niveles de abstracción es donde reside la fuerza de los BTs. Dado que todas las tareas tienen una interfaz en común y pueden contenerse unas en otras, se puede construir fácilmente una jerarquía de tareas que permita despreocuparse de los detalles de implementación de las sub-tareas.

En nuestro caso utilizamos los BTs para dictaminar el comportamiento de un bot en función de todo lo que ocurra en su entorno en ese momento. Básicamente, un BT está compuesto por varios nodos, que a su vez están formados por una condición y una acción. Como la condición de cada nodo puede cumplirse o no, tenemos simplemente 2 hijos para todos aquellos nodos que no sean una hoja.

Un nodo que no sea hoja puede no realizar ninguna acción, y servir simplemente para ramificar en un sentido u otro el comportamiento que deseamos que tenga el bot. A su vez, un nodo hoja puede no necesitar ninguna condición (sería siempre cierta), debido a que gracias a los nodos ancestros, se ha llegado hasta él y no necesita comprobar nada más. Por ejemplo, si queremos representar la anterior primera máquina de estados sencilla en forma de árbol de comportamiento, tendríamos lo siguiente:



Ejemplo 3: Representación del Ejemplo 1, realizado mediante un FSM, en un BT.

Puede parecer que este BT realiza lo mismo que una máquina de estados (en realidad se puede pasar de un árbol a una máquina de forma muy sencilla), pero la principal diferencia como ya hemos visto, es que con el árbol, los cálculos se realizan de forma mucho más rápida y más eficiente. A su vez, en este ejemplo también se puede apreciar la gran diferencia comentada anteriormente entre los BTs y los FSM, que es la ventaja que añaden modularidad en el código. Enemigo a la vista junto con moverse y disparar son una condición y dos acciones que no nos hace falta saber cómo están implementadas por debajo.

Para este proyecto, vamos a utilizar la implementación en JAVA de los árboles de comportamiento realizada por Ricardo Juan Palma Durán. La herramienta sobre la cual vamos a poder implementar en Java árboles de comportamiento está basada sobre el modelo de BT especificado en el libro "Artificial Intelligence for Games", segunda edición, escrito por Ian Millington y John Funge.

4.2.1. Java Behaviour Trees

Java Behaviour Trees (JBT) es un framework de Java para construir y ejecutar árboles de comportamiento en Java. Esta herramienta tiene dos ventajas, la primera y más importante es que es libre, pudiéndola utilizar sin restricción alguna, y la segunda es que se encuentra muy bien documentada.

JBT se compone de dos partes fundamentales. Por un lado está el JBT Core, donde se encuentran implementadas todas las clases que se necesitan para crear y ejecutar BTs. Básicamente, JBT Core permite al usuario crear BTs en Java y ejecutarlos después. JBT Core incluye diversas herramientas que automatizan este proceso de creación para facilitarnos la tarea de crear BTs. En particular, el JBT Core puede crear el código Java a partir de una descripción en formato XML, haciendo que el usuario solo se tenga que preocupar por la definición correcta de un BT en XML y la implementación a bajo nivel de las acciones y condiciones que utilizará el árbol. Esta implementación es dependiente del dominio, esto quiere decir que el código que se inserte dentro de las acciones y condiciones dependerá del juego que se esté desarrollando.

Por otro lado está el JBT Editor, el cual es una interfaz gráfica de usuario (GUI) que nos permite definir y diseñar BTs exportándolos después a un archivo XML que pueda comprender la unidad JBT Core. El JBT Editor ofrece un conjunto de nodos (tareas) estándar para crear árboles de comportamiento, de los cuales podemos destacar: nodo "Sequence", nodo "Selector", nodo "Dynamic Priority List", nodo "Subtree", etc. Permitiendo expandir este repertorio de tareas por las definidas por un usuario. Para poder llevar a cabo esta agregación de tareas, se han de definir en un fichero XML denominado MPPM (Make Me Play Me) para su posterior carga y visualización en el JBT Editor.

El MPPM es un dominio que permite definir el conjunto de condiciones y acciones que determinan el comportamiento de un bot de forma abstracta, es decir, sólo su nombre, nada de código.

Los JBT trabajan mediante un modelo conducido por *ticks*. La ejecución de un *tick* da al árbol tiempo para que actualice su estado y pueda ejecutarse de nuevo. Es importante diferenciar entre Modelo de árbol y Ejecución de un árbol. El modelo de un árbol es el

árbol en sí, es decir, la definición realizada anteriormente mediante el editor de JBTs, mientras que la ejecución se realiza sobre el modelo del árbol. De esta manera, es posible realizar varias ejecuciones simultáneas sobre un mismo modelo. Tras definir el modelo, es necesario crear una librería que lo incluya para poder ser utilizado.

Para poder ejecutar el modelo de un árbol se necesitan la librería que incluye ese modelo y un contexto de ejecución. El contexto de ejecución sirve para almacenar variables y valores que puedan necesitar los nodos para ir por un camino u otro del árbol. A partir de la librería, se puede extraer el modelo del árbol que va a ejecutarse, necesario para construir un ejecutor sobre él.

4.2.2. Pasos para construir en JBT un BT

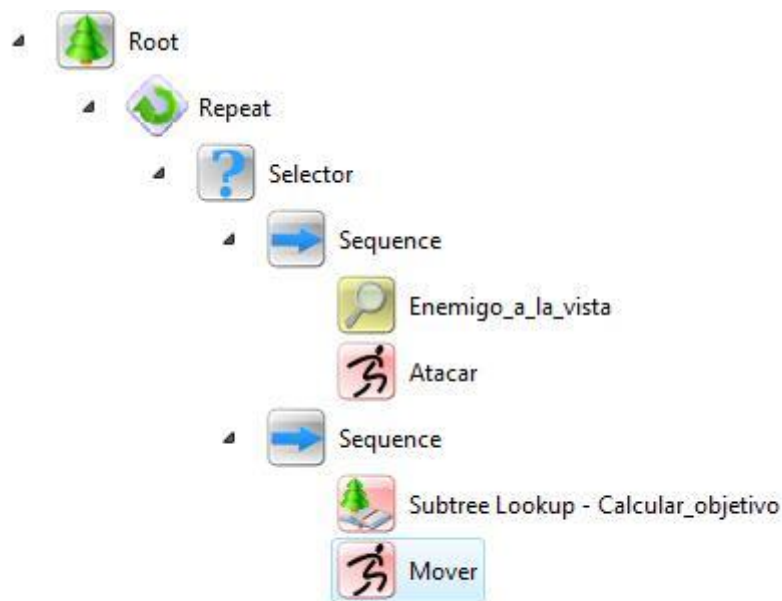
Los pasos a seguir para poder construir con la herramienta JBT un árbol de comportamiento son los siguientes:

- Definir las acciones y condiciones de más bajo nivel que usará el árbol. Estas acciones y condiciones serán definidas en el archivo MMPM.
- Crear el, o los modelos de BTs mediante el editor JBT Editor. Una vez finalizada la edición de los BTs, estos serán exportados en un fichero XML.
- Crear la declaración de los BTs especificados en el XML en Java. Esto se hace de forma automática mediante una herramienta que nos proporciona el propio JBT.
- Implementar las acciones y condiciones definidas en el MMPM. El usuario tiene que definir cada una de estas tareas básicas que podrá realizar el bot. JBT no sabe en qué dominio se encuentran estas tareas por ello el usuario tiene que implementarlas en Java mediante el código utilizado por el juego al que le está definiendo la IA.
- Ejecutar el modelo de BT mediante la utilización de las clases que nos proporciona el JBT Core

4.2.3. Ejemplo de un modelo de BT

A continuación veremos un ejemplo sencillo de un modelo de BT. En este árbol, estarán presentes como nodos hoja (tareas básicas a realizar por el bot) las acciones vistas anteriormente en el Ejemplo 3 (moverse y disparar), mientras que la condición (enemigo a la vista, también vista en el Ejemplo3) servirán para ramificar el árbol en un

sentido u otro o para filtrar la realización de unas acciones u otras. El árbol construido mediante la implementación de los JBT es el siguiente:



Ejemplo 4: Modelo sencillo de un BT.

Los nodos rojos con una figura humana dentro representan las acciones definidas anteriormente en el MMPM (Atacar y Mover). El nodo marrón con una lupa dentro representa la condición definida (Enemigo_a_la_vista). Otros nodos remarcables en este ejemplo son el nodo “Selector”, el nodo “Sequence” y el nodo “Subtree Lookup”.

El nodo Selector evalúa de forma secuencial las tareas que tiene como hijos. La primera que se cumpla es la que se ejecuta descartando todas las siguientes. Con que una de las tareas que tiene el Selector se cumpla, éste tiene éxito.

El nodo Sequence realiza lo que indica, la secuencia de todas y cada una de las acciones que haya debajo. Si alguna de las acciones falla, la secuencia falla por completo.

El nodo Subtree Lookup – Calcular_objetivo sirve para ejecutar un sub-árbol definido en otro archivo. Es un nodo útil para no sobrecargar un árbol demasiado, tomando un conjunto de nodos como un nuevo árbol y definiéndolo aparte.

Aunque no aparezca en el ejemplo, otros nodos importantes a tener en cuenta son:

El nodo “Dynamic Priority List”. Este nodo sirve para identificar qué acción llevar a cabo de un conjunto de acciones dadas, con prioridad mayor a menor según estén más

arriba o más abajo en el árbol. Para ello, evalúa todas las condiciones (en este caso las condiciones que ejecuta son denominadas guardas) y para la primera que se resuelva como cierta, ordena ejecutar su nodo correspondiente. Cada cierto tiempo vuelve a evaluar todas las condiciones y si se hace cierta una de mayor prioridad, interrumpe la acción en curso y pasa a realizar la mayor prioritaria.

El nodo “Static Priority List”. Este nodo tiene la misma función que el Dynamic Priority List con la diferencia que el nodo Static no vuelve a evaluar cada cierto tiempo las condiciones de sus hijos, limitándose a realizar dicha evaluación al inicio una única vez

El nodo “Parallel”. Este nodo actúa en función de una política, pudiendo ser similar a un nodo Selector o a un nodo Sequence. Esta política será definida por el usuario cuando cree este tipo de nodos. La diferencia con respecto a estos nodos radica en la forma de ejecutar las tareas hijas. Tanto el Selector como el Sequence evalúan y ejecutan sus tareas de forma secuencial mientras que el Parallel lo hace de forma concurrente, ejecutando a la par todos sus nodos hijos.

5. Unreal Tournament 2004



Ilustración 1.

Unreal Tournament 2004, también conocido como UT2004, es un videojuego de acción en primera persona (FPS), desarrollado por Epic Games y Digital Extremes y publicado por Atari. UT2004 es parte de la serie Unreal y es la secuela de Unreal Tournament 2003, remplazando a éste completamente dado que UT2004 trae todo el contenido de UT2003. Esta serie está orientada a la experiencia multijugador aunque incluye un modo de un solo jugador que emula el juego multijugador a través del uso de entidades controladas por el ordenador, que simulan a los jugadores humanos, llamadas bots.



Ilustración 2.

La mayor adición que UT2004 hace al universo Unreal es la introducción de vehículos a la fórmula de los FPS. Esto unido con la posibilidad de modificar mapas, armas y modos de juego (llamados mods) gracias a la herramienta “UnrealED” (incluida en el juego) hace que el mundo Unreal Tournament y en especial UT2004 sea todo un éxito.



Ilustración 3.



Ilustración 4.

6. Pogamut

Nuestro proyecto consta, a grandes rasgos, de la programación de una IA destinada a un bot del juego UT2004. Para ello necesitamos utilizar una herramienta que nos permita interactuar con el juego, proporcionándonos una capa de abstracción para centrarnos única y exclusivamente en la programación de la IA del bot. Esta herramienta es Pogamut.

Pogamut es una herramienta que se encuentra en continuo desarrollo por un equipo perteneciente a la facultad “Charles University” de Praga. Actualmente se encuentra en la versión 3.2.3 aunque en nuestro proyecto hemos utilizado la versión 3.0.11.

Pogamut nos permite desarrollar de forma sencilla diferentes comportamientos que pueden adoptar los bots del UT2004. Esto es posible gracias a que el motor de juego del UT2004 está abierto, mediante script (UnrealScript), para poder modificar la IA de los bots. Pogamut utiliza UnrealScript y proporciona un plugin para la plataforma NetBeans de modo que los bots pueden ser programados en lenguaje Java.

La herramienta está pensada para ofrecer la posibilidad de programar inteligencias artificiales de manera cómoda, simplificando la parte “física” de la creación del bot (cálculo de caminos, conexión y obtención de información del entorno) en unos pocos comandos, de manera que el programador se pueda centrar en las partes más interesantes competentes a la creación de una IA.

Pogamut se desarrolla principalmente para dos fines bien definidos. El primero se centra en el uso de esta herramienta para enseñar a los alumnos los posibles problemas que tiene el desarrollo de agentes virtuales. El segundo fin está destinado para la investigación y la evaluación de distintos comportamientos que puedan adoptar los bots. Estos fines unidos a una buena documentación, a librerías para crear bots tanto en UT2004 como en Unreal Engine 2 Runtime (UE2R, plataforma estable y robusta para tutoriales interactivos y simulaciones), mapas de ciudades creados para UE2R y el plugin para NetBeans, que nos permite un perfecto control de la ejecución de nuestros bots pudiendo tener una visualización en 3D de estos en cada uno de los mapas (como podemos observar en la Imagen 1), hace que Pogamut tenga un gran peso en el ámbito del aprendizaje. Estos son unos de los motivos que nos invitaron a utilizar esta herramienta en nuestro proyecto.

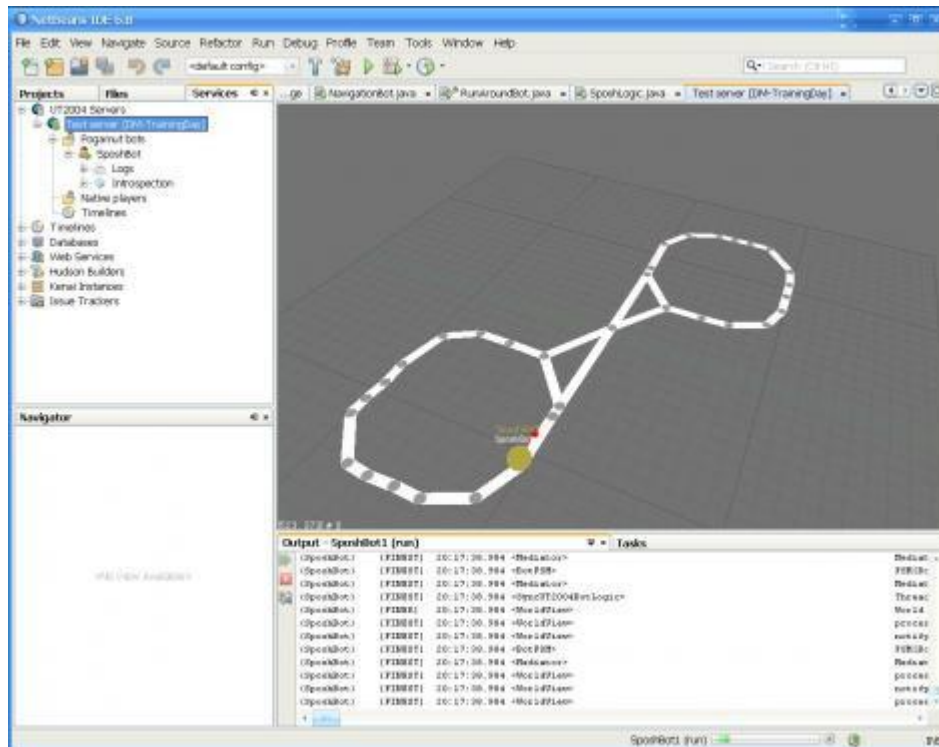


Imagen 1: Visualización del mapa y la posición de los Bots en tiempo real.

Para poder utilizar la herramienta Pogamut (versión 3.0.11) es necesario disponer de la versión 6.8 o posterior del Netbeans, la versión 1.6 o superior del JDK, una copia del UT2004 o descargarse el UE2R (dado que Pogamut nos sirve para experimentar con IAs tanto en UT2004 como en UE2R. En nuestro caso nos olvidaremos de UE2R pese a que sea una interesante aplicación y nos centraremos en construir bots destinados a UT2004). Es importante remarcar que para que el Pogamut interactúe correctamente con el juego, éste debe ser instalado en inglés. Además, debe estar actualizado a la versión 3669 (incluida en el instalador de la herramienta).

Una vez descargada la última versión del Pogamut, simplemente hay que seguir las instrucciones que aparezcan en pantalla para instalarlo, indicando el lugar donde se quiere instalar la herramienta y los lugares donde están instalados tanto el Netbeans como el juego UT2004 y el UE2R. Al acabar, iniciando el Netbeans se instalarán los plugins necesarios para que la herramienta funcione.

6.1. Como conectar el Pogamut con el UT2004

La siguiente imagen (Imagen2), esboza la arquitectura que implementa por debajo el Pogamut.

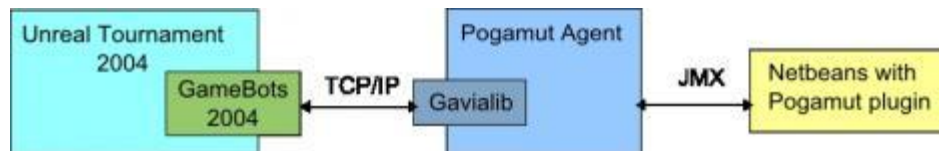


Imagen 2: Arquitectura de la conexión entre el UT2004 y el Pogamut.

La información correspondiente al UT2004 es exportada mediante TCP/IP utilizando el protocolo de texto GameBots2004. Estos mensajes son entendidos y procesados por la librería Java GaviaLib, de manera que el agente (bot) construido con Pogamut pueda trabajar con objetos Java. Una vez interpretados y manipulados estos mensajes por el bot, éste devuelve en forma de comandos el resultado al juego mediante la conexión TCP/IP. Este agente puede a su vez ser depurado remotamente mediante el protocolo JMX a través el plugin de Netbeans.

Como se comentó antes a grandes rasgos, UT2004 proporciona mediante UnrealScript una interfaz (API) hacia el exterior del UnrealEngine, quien es el encargado de ejecutar todo el entorno del UT2004. A esta parte de la plataforma que se encuentra programada por UnrealScript se le denomina GameBots2004 (GB2004).

GB2004 actúa como un servidor TCP/IP. Éste define un protocolo de texto que tiene que ser implementado para poder conseguir que el Cliente pueda ejecutar de forma efectiva el bot en el entorno del UT2004. Este protocolo consiste en el envío de mensajes por parte del GB2004 al Cliente y de comandos por parte del Cliente al GB2004. Los mensajes sirven para dar información y conocimiento al Cliente periódicamente sobre los eventos (mediante mensajes asíncronos) o estados (mediante mensajes síncronos) ocurridos en el juego. Mientras que los comandos son usados para controlar a cada uno de los bots que se están ejecutando en el servidor en función de los mensajes recibidos.

Cuando el cliente se conecta con el GB2004, éste envía “HELLO” al cliente. El cliente debe responder con un “READY”. GB2004 envía la información sobre el juego y el mapa que UT2004 está ejecutando. A su vez envía los puntos de navegación (NavPoints) y objetos que contiene el mapa en cuestión. Después de esta interacción,

el GB2004 se queda esperando a que el cliente envíe un “INIT”. Cuando lo recibe, el juego genera los bots y habilita la comunicación entre el GB2004 y el Cliente para comenzar con los envíos de mensajes con información del entorno y recepción de los comandos que envíen los bots.

GaviaLib es el encargado de traducir los mensajes recibidos mediante TCP/IP del GB2004 a objetos java para que el bot pueda interpretar dichos mensajes. A su vez se encarga de codificar los comandos que el bot envía hacia el GB2004 para que éste los comprenda. A groso modo, podemos definir el GaviaLib como un traductor entre el Pogamut y el UT2004. En la siguiente imagen (Imagen 3) proporcionada por la web Pogamut-Devel, podemos ver la arquitectura del GaviaLib.

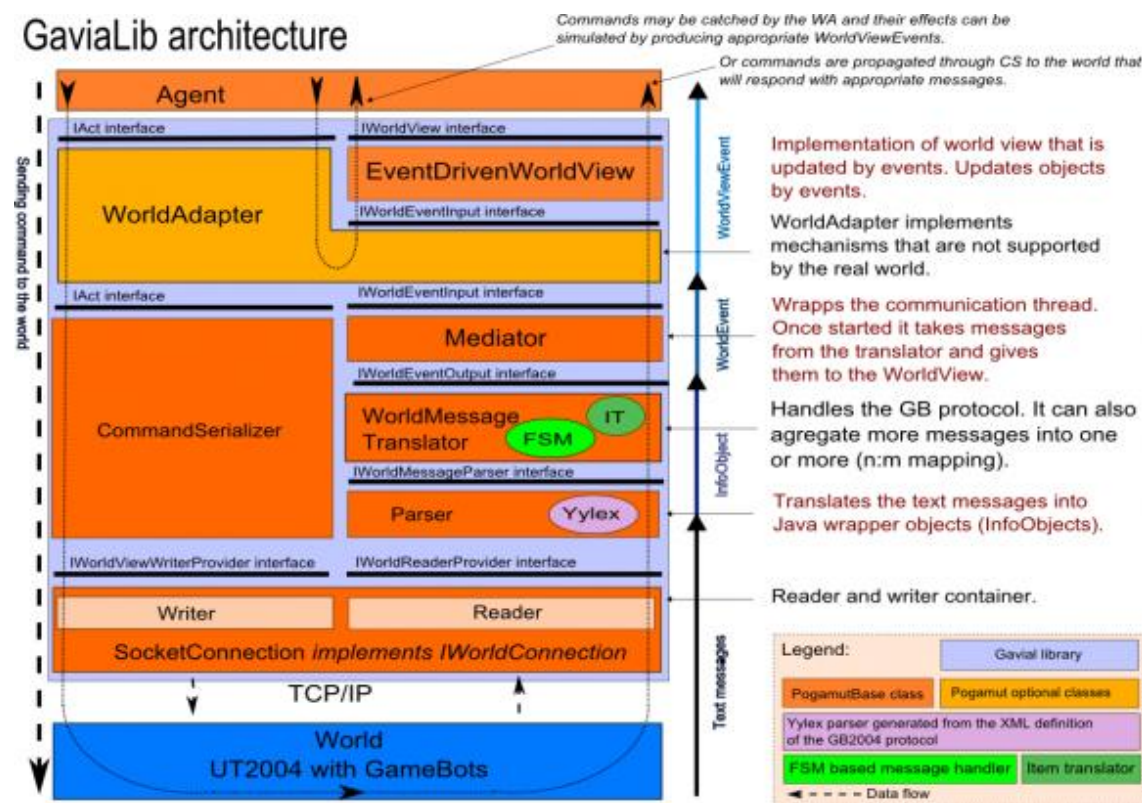


Imagen 3: Arquitectura del GaviaLib.

Finalmente, gracias al protocolo JMX podemos, a través del IDE (NetBeans), crear los agentes programados con el Pogamut en el entorno que nos proporciona UT2004, controlarlos y depurarlos en tiempo real (cuando estos están siendo ejecutados por UT2004). Pudiendo observar en todo momento sus parámetros y las variaciones de éstos.

7. Un Bot en Pogamut

Al crearse un bot en Pogamut, se crean una serie de módulos (variables instancia) que resultan de utilidad para conocer información del bot, de su entorno, de la partida, para calcular el camino entre 2 nodos del mapa, etc. A continuación vamos a explicar los módulos más remarcables que utilizamos en la programación de nuestro bot.

7.1. Módulos básicos de un Bot

La variable “world”

Se encarga de la visión global del mundo de la partida, permitiendo obtener todos los objetos de un tipo en concreto, ya sean jugadores, nodos de navegación o armas. También permite registrar Listeners sobre objetos o sobre eventos.

La variable “info”

Mantiene la información referente al estado interno del bot: salud, adrenalina, velocidad de movimiento, equipo (en partidas por equipos), munición para el arma en uso, etc. Permite a su vez obtener el objeto más cercano al bot en ese momento, siendo objeto un punto de navegación, un arma/munición o un jugador, ya sean visibles o no por el bot.

La variable “bot”

Representa la instancia del bot controlado en ese momento y guarda la información de todo aquello que relaciona al bot con el juego, ya sea su posición en el mapa, su nombre o incluso el controlador que lo mantiene relacionado con el Pogamut.

La variable “weaponry”

Esta variable, como su propio nombre indica, guarda toda la información sobre las armas que en ese momento posee el bot. También permite acceder a la cantidad de munición principal y secundaria para cada una de ellas.

La variable “pathPlanner”

Una de las variables más importantes dentro del bot es ésta, el planificador de caminos. Nos permite calcular el camino entre dos posiciones del mapa a través de la red de caminos que se extiende entre cada nodo de navegación (Ver Imagen 4). El

único método importante que posee esta variable es el método 'computePath', que recibe una localización objetivo del camino y devuelve la serie de nodos de navegación que forman el camino desde la posición actual y la destino indicada por parámetro.



Imagen 4: Red de caminos generada para el mapa Chrome.

La variable “pathExecutor”

La variable que sigue en importancia al planificador de caminos es el ejecutor de caminos. Esta variable es la que se encarga de ejecutar el camino calculado por el planificador, es decir, ordena al bot a qué nodo de la secuencia debe ir para continuar con el camino. Esta variable posee a su vez métodos para acceder a un nodo en concreto del camino, al anterior por el que se pasó o al siguiente por el que se pasará. También da la oportunidad de ordenar al bot que retroceda el número de nodos que se desee en el camino.

La variable “game”

Este módulo se encarga de guardar toda la información referente a la partida en curso, incluyendo el nombre del mapa y el tipo de partida, la salud o armadura máximas que puede alcanzar un jugador, el tiempo restante que queda para que la partida finalice y

la puntuación límite dependiendo del tipo de juego: en una partida deathmatch será el número de muertes, mientras que en una de capturar la bandera será la puntuación total establecida.

La variable “players”

Otra variable importante es la variable players. Mantiene la información de todos los jugadores actualmente en la partida y permite al bot saber cuáles son amigos o enemigos. Así mismo, facilita la obtención del amigo o enemigo más cercano a la posición en la que se encuentre al bot o simplemente si se quiere saber si el bot ve a algún enemigo, sin necesidad de que sea el más cercano (puede que el más cercano se encuentre tras una pared, pero en ese caso, el bot no lo vería).

La variable “senses”

Esta variable recoge toda la información de los “sentidos” artificiales del bot. Esta información incluye la forma en que el bot murió (por una caída, un disparo, etc.) o si oye un ruido de pasos. Es útil porque permite percibir cuándo están atacando al bot por detrás para poder actuar en consecuencia.

La variable “body”

Esta variable envuelve de forma sencilla distintos módulos de comando como Action, AdvancedLocomotion, AdvancedShooting, Communication y SimpleRayCasting para poderlos controlar. El que nos interesa es el comando de control del SimpleRayCasting para poder manejar éste a nuestro antojo. El SimpleRayCasting es un módulo que lanza un rayo que nos detecta si existe alguna colisión con algún objeto del mundo, enviando dicha información, la cual es capturada por el Listener correspondiente para su posterior tratamiento.

7.2. Estructuras básicas de un Bot

Todo bot programado mediante la plataforma Pogamut se compone básicamente de 6 métodos, 4 de los cuales son llamados al ejecutarse el bot por primera vez. Otro método se llama iterativamente cada cierto tiempo y es el que controla la inteligencia del bot, mientras que el último se ejecuta una vez muere el bot.

Los siguientes 4 métodos son llamados secuencialmente una única vez por el motor del Pogamut tras ordenar al bot que empiece a funcionar mediante el método startAgent().

7.2.1. Método “prepareBot”

Este método se ejecuta antes de que el bot se conecte al juego y sirve para inicializar los módulos del bot, por ejemplo el ejecutor de caminos. También es el lugar ideal para inicializar los atributos internos del bot.

7.2.2. Método “getInitializeCommand”

Este método, ejecutado inmediatamente después del anterior, se encarga de definir los atributos del bot “importantes” para el Pogamut, es decir, el nombre, la posición de inicio en el mapa, el equipo del que forma parte en una partida de juego por equipos (capturar la bandera por ejemplo), etc.

Para ello, dentro del método se crea un objeto Initialize y luego se configura todo aquello remarcable del bot (nombre, etc. como se mencionó antes), que será el que devuelva el método.

No es necesario dar valor a todos ellos, simplemente a los que tengan relevancia en el comportamiento o definición de cada bot; el Pogamut se encarga de dar a los no definidos un valor por defecto.

7.2.3. Método “botInitialized”

El tercer método en la secuencia se ejecuta tras recibir en bot el mensaje INITED desde el servidor. Esto indica que el comando INIT enviado por el bot para conectarse al servidor no falló y se ha creado un enlace entre el bot y el servidor por el cual el bot puede recibir nuevos mensajes. No significa que el bot ya haya aparecido en la partida, simplemente que la conexión entre él y el servidor se ha realizado sin problema alguno.

El método posee 3 parámetros que recibe una vez es llamado: la información del juego en el que el bot va a tomar parte (tipo de juego, mapa, etc.), la configuración actual del bot (incluyendo lo definido en el método ‘getInitializedCommand()’) y las variables propias del bot tales como velocidad de movimiento, salud máxima, etc.

En este método se pueden definir las características de otros módulos del bot, así como atributos que no se definieron en el método anterior y que era necesario esperar a este momento, en que el bot ha establecido la conexión con el servidor, para configurarlo en consecuencia.

7.2.4. Método “botSpawned”

El último método de la secuencia se ejecuta una vez que el bot ha aparecido en el mapa por primera vez. Se recibe un parámetro llamado “self” (sí mismo), que recoge la información actual del bot referente a su localización en el mapa y su actual estado.

Este método es el último momento en el que se pueden realizar tareas de preparación antes que el método logic sea llamado periódicamente.

7.2.5. Método “logic”

A continuación, el motor del Pogamut se encarga de llamar periódicamente al método “logic()”, unas 4 veces cada segundo. Es el método más importante, ya que es el que emula la inteligencia artificial del bot. Mediante este método se define el comportamiento del bot que no se haya definido en cualquiera de los otros módulos, ya sea mediante una máquina de estados o mediante otro método.

El hecho de ser llamado una vez cada cuarto de segundo permite simular de manera bastante efectiva una inteligencia. Facilita al bot la capacidad de tomar decisiones muy rápido basándose en los cambios que sucedan en el juego, ya sea ir a buscar vida, entrar en combate si ve a un enemigo o perseguir a un adversario que haya desaparecido del campo de vista del bot.

Esta rapidez de cambio de estrategia y de toma de decisiones permite a un bot bien programado simular el comportamiento de un jugador humano que podrá engañar fácilmente a jugadores poco avezados en estos tipos de juegos. De todos modos, aunque la rapidez de cálculo de posibilidades y toma de decisiones sea notable, no es comparable a la que un humano pueda llevar a cabo en la realidad y mucho menos semejante a la que un jugador experimentado pueda poseer.

7.2.6. Método “botKilled”

El último método de la estructura básica de un bot es el método “botKilled()”. Este método se ejecuta una vez que el bot muere en la partida y resulta de utilidad para reiniciar variables internas del bot y dejarlo “como nuevo” para que la próxima vez que vuelva a ser ejecutado el método logic, sea como si fuera la primera vez que el bot apareció en el juego.

8. Comunicación entre Bots

Vamos a orientar nuestro proyecto al modo de juego Capturar la Bandera, un modo de juego cooperativo que consiste, como su propio nombre indica, en capturar la bandera del equipo contrario y llevarla a la base del propio para anotar un punto. Como este es un modo de juego que se basa sobre todo en la cooperación, necesitamos un mecanismo para comunicar a los bots de cada equipo entre sí. Utilizamos una estructura de buzones para realizarlo.

8.1. Buzones

Cada buzón está construido utilizando el patrón Singleton, de manera que todos los bots puedan acceder al mismo buzón. El acceso a los buzones está sincronizado, de manera que no pueda haber 2 bots accediendo a la vez a un mismo recurso. Este bloqueo/desbloqueo lo realizamos mediante 2 métodos:

```
public synchronized boolean bloqueaBuzon()
{
    if (bloqueado == false)
    {
        bloqueado = true;
        return false;
    }
    return bloqueado;
}

public void liberaBuzon()
{
    bloqueado = false;
}
```

Imagen 5: Métodos necesarios para implementar un buzón.

El acceso sincronizado al bloqueo del buzón nos garantiza que no haya 2 intentos de bloqueo simultáneos. Por su parte, cada bot realiza una espera activa mediante un bucle sobre el método `bloqueaBuzon()` hasta que recibe que el buzón se ha liberado, momento en el que entra al método y lo bloquea para sí.

Para comunicar a los bots, poseemos 3 buzones:

- Primeramente, tenemos un `BuzonGeneral` que simplemente lo que hace es distribuir todos los bots que se van creando uniformemente entre los dos equipos disponibles (rojo y azul) mediante un atributo booleano que cambia cada vez que un bot es asignado a un equipo.
- Un buzón para el equipo rojo y otro para el equipo azul. Estos buzones son los encargados de almacenar toda la información que permite a los bots relacionarse entre sí. Esta información incluye si la bandera del equipo ha sido cogida o no y las ayudas que han pedido otros bots al encontrarse con resistencia. Estos buzones también sirven para distribuir a los bots dentro de un equipo entre los bots de ataque que deben capturar la bandera contraria y los bots de defensa que deben proteger la propia. Dentro de los bots de defensa, también distribuye entre los que se quedan al lado de la bandera defendiéndola o los que patrullan por una zona cercana a la bandera en busca de enemigos.

9. Nuestro Bot

Como hemos mencionado antes, nuestro bot se comportará de acuerdo a un árbol de comportamiento implementado mediante los JBT, pero estamos limitados por la herramienta Pogamut para comunicarnos con el UT2004. Por esto, necesitamos interconectar ambos sistemas de manera que los árboles funcionen bajo Pogamut.

Para llevarlo a cabo hemos tenido que “hacer creer” al Pogamut que la implementación de los árboles pertenece al proyecto Pogamut en el que está programado el bot. Para esto, hemos tenido que copiar los archivos fuente generados para cada acción y condición definidos en el dominio MMPM y la librería que incluye la definición del árbol de comportamiento en la carpeta de los archivos fuente del bot.

9.1. La estrategia

El grupo se divide en dos subgrupos iguales, uno de ataque y otro de defensa. En el caso que sean impares un miembro más en el subgrupo de defensa que en el de ataque. Todos los jugadores mantienen su rol durante toda la partida.

Capturar bandera contraria.

Cada bot se mueve independiente a sus compañeros. Con el fin de que cada uno de ellos consiga la bandera. Comunican si en algún momento en su camino se encuentra alguna hostilidad. En el caso de que sean 2 o más enemigos, se pide ayuda. Cada bot comprueba si hay peticiones de ayuda y en caso de que se encuentre cerca responde.

NOTA: los bots de ataque sólo se encargan de sus conflictos, pasan de los conflictos que tengan los bots de defensa

Proteger bandera contraria.

Una vez que uno de los bots de ataque consiga la bandera, todos los restantes se reagrupan con éste y lo cubren. El portador calcula la ruta de vuelta a su base. Si se encuentran un conflicto TODOS se involucran.

Si la bandera se cae en el transcurso de llevarla a la base, los bots de ataque cambian a estado Capturar bandera.

Defender bandera propia.

Si el grupo de defensa tiene a 2 jugadores o menos, se quedan defendiendo al lado de la bandera. A partir de 3, la diferencia se dispersa de manera independiente por su base. Cuando un bot del grupo de defensa que patrulla se encuentra con un enemigo, si estos son 2 o más, pide ayuda a la patrulla. Si el conflicto se produce involucrando a uno de los dos bots que defienden la bandera toda la patrulla que no se encuentre en combate asiste a estos dos bots. Los que se encuentren en combate terminan y después apoyan.

NOTA: pueden dar apoyo a bots de ataque con problemas que se encuentren cerca de la base.

Recuperar la bandera.

Todo el grupo de defensa persigue al que lleva su bandera. Si la bandera cae se sigue manteniendo el estado de recuperar.

9.2. El árbol de decisiones

Para el árbol de decisión de estrategia (capturar, defender, etc.), primero necesitamos definir las acciones y condiciones básicas que necesitaremos en el árbol. Definimos el siguiente dominio MMPM (ver Imagen 6)

Cada acción y condición se explicarán posteriormente con más detalle. Hay que decir que por cada acción y condición definidas aquí, hay un método correspondiente en el código del bot, que será el encargado de realizar lo que deba de acuerdo a la acción/condición oportuna.

Una vez definido este dominio de acciones y condiciones, pasamos a la creación de los árboles que regirán el comportamiento del bot en cada situación. Debemos recalcar que todos los bots utilizan el mismo modelo de árbol, con lo que las distinciones entre equipos o entre grupo de ataque o defensa tienen que hacerse dentro de cada acción y cada condición.

```

<ActionSet>
  <Action name="capturarBandera"/>
  <Action name="protegerBandera"/>
  <Action name="defenderBandera"/>
  <Action name="recuperarBandera"/>
  <Action name="ayudarCompaniero"/>
</ActionSet>
<SensorSet>
  <Sensor name="portarBandera" type="BOOLEAN"/>
  <Sensor name="noPortarBandera" type="BOOLEAN"/>
  <Sensor name="banderaEnBase" type="BOOLEAN"/>
  <Sensor name="noBanderaEnBase" type="BOOLEAN"/>
  <Sensor name="ayudaAtaque" type="BOOLEAN"/>
  <Sensor name="ayudaPatrulla" type="BOOLEAN"/>
  <Sensor name="ayudaBase" type="BOOLEAN"/>
  <Sensor name="companieroCerca" type="BOOLEAN"/>
</SensorSet>

```

Imagen 6: Contenido del fichero MMPM de la estrategia.

9.2.1. Árbol Madre

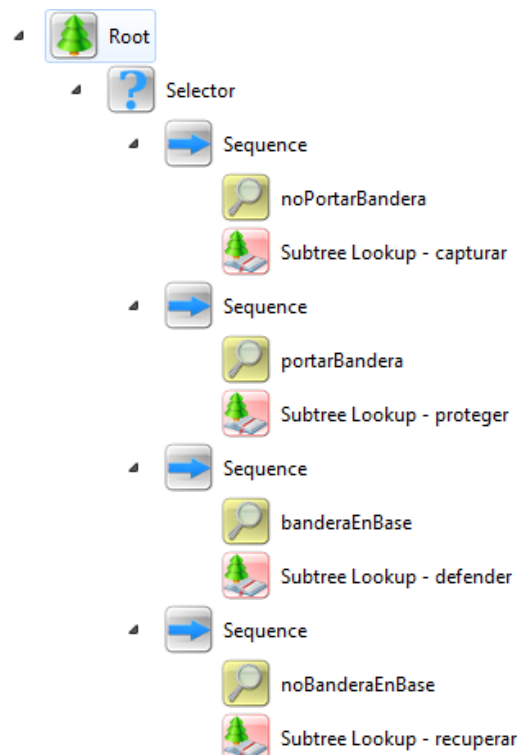


Imagen 7: Árbol Madre.

El árbol madre se compone de un nodo Selector como nodo raíz que contiene a 4 nodos Sequence uno para cada una de las tareas que componen la estrategia. Cada nodo Sequence contiene una condición y una acción (sub-árboles en realidad) que serán evaluadas de forma secuencial.

Para la secuencia que contiene al nodo capturar bandera, su condición (*condición noPortarBandera*) es que la bandera enemiga no la esté portando ningún bot del equipo. Esto se comprueba accediendo al estado de la bandera contraria y observando si se encuentra en “casa” o ha caído. La condición del nodo proteger es la análoga a la del nodo anterior: si alguien porta la bandera, hay que protegerle (*condición portarBandera*).

Con respecto a los nodos defender y recuperar, sus condiciones (*banderaEnBase, noBanderaEnBase*) son análogas la una con la otra al igual que las de los 2 nodos recientemente mencionados. Cuando a bandera se encuentra en la base, se ejecuta el nodo defender, mientras que si un bot del equipo contrario ha cogido la bandera propia, ésta no se encuentra en la base y hay que recuperarla.

Otro aspecto importante de este árbol y sus condiciones es que los bot de ataque sólo comprobarán las 2 primeras (para las de defender y recuperar devolverán falso), mientras que los bots de defensa, por el contrario, devolverán falso en las dos primeras y comprobarán las 2 segundas.

9.2.2. Árbol Capturar

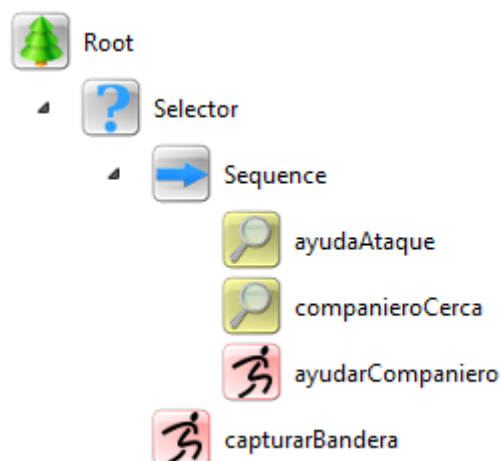


Imagen 8: Árbol Capturar.

Este árbol se compone de un selector de 2 nodos: una secuencia correspondiente a una ayuda que pueda brindar el bot a un compañero y otro nodo correspondiente a la acción de capturar la bandera en sí (*acción capturarBandera*). El nodo selector se comporta, como se dijo anteriormente, como un if-then-else, de manera que si la secuencia de ayuda falla, se ejecute la acción de capturar bandera.

En esta secuencia de ayuda, primero se comprueba que exista alguna petición de ayuda de un bot de ataque (*condición ayudaAtaque*) mirando el vector de ayudas de ataque del buzón de equipo correspondiente. Si existe alguna petición de ayuda, se comprueba si el compañero está cerca o no (*condición compañeroCerca*), que para un bot de ataque se traduce en que el compañero esté más cerca del bot que de la bandera contraria; en caso de que haya algún bot que cumpla este requisito, el método que ejecuta esta condición se guarda el lugar donde se requiere la ayuda. En caso de que la condición anterior no falle, simplemente se calcula la ruta hasta el lugar del compañero y se ejecuta el movimiento (*acción ayudarCompañero*).

La acción de capturar la bandera simplemente calcula el camino hasta donde se encuentra la bandera (ya sea en la base o en el lugar donde ha caído tras ser derrotado el portador) y realiza el movimiento por ese camino.

9.2.3. Árbol Proteger

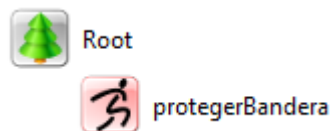


Imagen 9: Árbol Proteger.

El árbol correspondiente a la orden de proteger la bandera consiste en simplemente una acción (*protegerBandera*).

Cada bot posee un atributo portador de manera que en todo momento sepa quién es el portador de la bandera contraria.

Cuando el bot que coge la bandera entra por vez primera en el método correspondiente al nodo *proteger_bandera*, introduce en el buzón de su equipo su identificador único para que los demás del equipo puedan acceder a él y seguirle. Una vez que ha hecho esto, comienza su vuelta a la base para entregar la bandera.

Los demás bots del equipo de ataque, acceden al buzón y guardan en su variable portador el identificador colocado anteriormente. Lo primero que hacen es replegarse

a una posición intermedia del mapa para luego comenzar a seguir al portador de la bandera hasta la base.

Es importante recalcar que para el bot que lleva la bandera, su atributo portador siempre será null, con lo que hay que restringir su entrada en los métodos que accedan a tal variable.

9.2.4. Árbol Defender

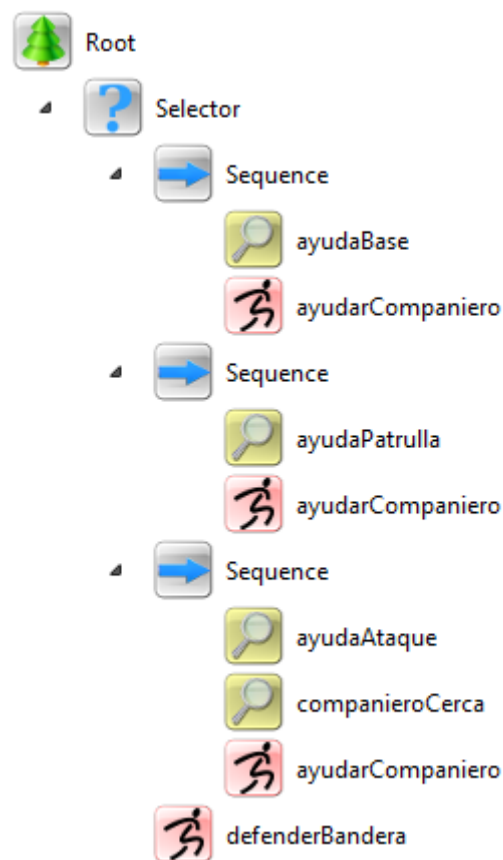


Imagen 10: Árbol Defender.

El árbol de defensa se compone principalmente de un nodo Selector, al igual que el de capturar. Cada rama del nodo selector está formada por las secuencias de comprobaciones de ayuda que deben hacer los bots de defensa y un último nodo que establece el comportamiento por defecto de los bots al defender (*acción defenderBandera*).

La primera secuencia trata sobre las ayudas que hayan podido pedir los bots que se quedan al lado de la bandera defendiéndola. Este nodo es ejecutado por todos los bots

de defensa, tanto si pertenecen a la defensa de la bandera o a la patrulla. Primero se comprueba si hay alguna petición de ayuda de un bot de la base (*condición ayudaBase*) y en caso de que exista, todos los bots de defensa se dirigen a la zona para ayudar (*acción ayudarCompaniero*).

La segunda secuencia es ejecutada exclusivamente por los bots de la patrulla y se encargar de observar si algún otro bot de la patrulla ha solicitado ayuda o no (*condición ayudaPatrulla*). En caso de que así sea, se realiza el cálculo del camino hasta el lugar del conflicto (*acción ayudarCompaniero*), guardado anteriormente en la ejecución de *ayudaPatrulla*.

La tercera secuencia es ejecutada únicamente también por los bots de patrulla y realiza las mismas tareas que la secuencia de ayuda de los bots de ataque. Primero observa si un bot de ataque ha pedido ayuda (*condición ayudaAtaque*), si hay ayudas pendientes, comprueba si se encuentra cerca (*condición companieroCerca*) y en caso de ser así, se dirige al lugar (*acción ayudarCompaniero*). La diferencia con los bots de ataque es que la comprobación de si el compañero se encuentra cerca o no se hace tomando la distancia a la bandera propia en lugar de a la bandera del equipo contrario.

Finalmente, si no se cumple ninguna de las anteriores condiciones y fallan todas las secuencias, se realiza la defensa de la bandera. Para los bots de defensa de la base, esta acción consiste en ir a un lugar próximo a la bandera y esperar observando si llega algún enemigo. Para los bots de patrulla, consiste en dirigirse al primer lugar de su ruta de patrulla.

9.2.5. Árbol Recuperar

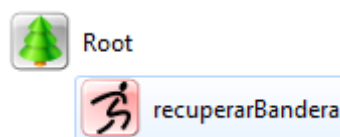


Imagen 11: Árbol Recuperar.

El árbol correspondiente a la orden de recuperar la bandera consiste en simplemente una acción (*recuperarBandera*). Esta acción, ejecutada por todos los bots de defensa, pertenezcan al grupo que pertenezcan, se basa en perseguir al portador de la bandera propia para darle caza y llevarla de vuelta a la base. Para ello cada bot del equipo de defensa accede al buzón del equipo contrario y obtiene el portador (portador de su propia bandera).

9.3. El árbol de acciones

El primer árbol nos sirve para decidir qué debe hacer el bot en cada momento. Pero las acciones como atacar o recoger salud no quedan contempladas en este árbol, con lo que construimos uno nuevo que controle estas acciones básicas.

Para este nuevo árbol definimos el siguiente domino MPPM:

```
<ActionSet>
  <Action name="Attack" />
  <Action name="Stop shooting" />
  <Action name="Rotate" />
  <Action name="Pursue" />
  <Action name="Get Health" />
  <Action name="Get Items" />
</ActionSet>
<SensorSet>
  <Sensor name="Must attack" type="BOOLEAN" />
  <Sensor name="Must stop shoot" type="BOOLEAN" />
  <Sensor name="Is being shot" type="BOOLEAN" />
  <Sensor name="Must pursue" type="BOOLEAN" />
  <Sensor name="Must get health" type="BOOLEAN" />
  <Sensor name="Can get items" type="BOOLEAN" />
</SensorSet>
```

Imagen 12: Contenido del fichero MPPM de las acciones.

Como se puede observar, existe una condición por cada acción que se puede realizar. Estas condiciones servirán de guardas para las acciones al construir el árbol. Obteniendo el siguiente árbol de comportamiento:

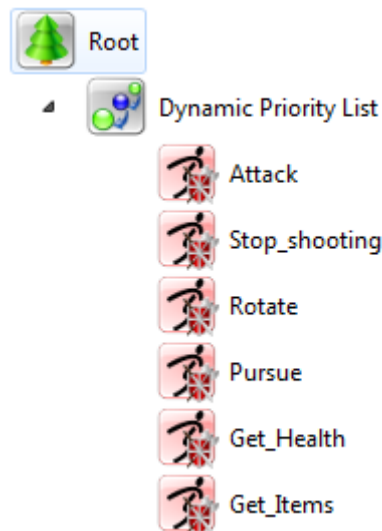


Imagen 13: Árbol de acciones.

Nuestro árbol de acciones sencillas se compone únicamente de una lista de prioridad dinámica de la que “cuelgan” todas las acciones. Como explicamos antes, la lista de prioridad dinámica es útil porque ejecuta la acción más prioritaria cuya guarda se haga cierta. De este modo, en este árbol, lo más prioritario será siempre atacar a un enemigo visible, mientras que la recogida de ítems queda en el último lugar de importancia. A continuación vamos a explicar el árbol en detalle.

9.3.1. Atacar

Lo primero y más prioritario siempre será atacar a un enemigo que se encuentre en el campo de visión del bot (*acción Attack*). Para ello, al ejecutarse la guarda de la acción (*condición Must_attack*), lo primero que realiza el bot es mirar si hay algún enemigo a la vista (mediante la variable “players”) y si tiene munición en algún arma (variable “weaponry”). Si ésto se cumple, entonces es viable atacar al enemigo. La acción de ataque consiste en varios pasos:

- En caso de no tener un enemigo asignado anteriormente, primero escoge un enemigo de todos los que puede ver (si sólo ve uno, pues será ése el elegido).
- A continuación mira si el arma que lleva equipada tiene munición. En caso de no tener, cambia a otra del inventario que sí tenga.
- Tras esto, comienza a disparar el enemigo.
- Por último, si la distancia entre ambos es relativamente grande, el bot corre hacia el enemigo.

A su vez, si al mirar si hay enemigos a la vista encuentra en su campo de visión a dos o más enemigos, el bot se encarga de realizar una petición de ayuda agregándola a uno de los tres vectores destinados a ese fin que están implementados en el buzón del equipo al que pertenece. Dependiendo que rol haya tomado el bot que pide la ayuda (ataque, patrulla o defensa), introducirá la petición en el vector correspondiente para que sea atendida por otro agente y este pueda ayudarlo.

Hay que recalcar que mientras el bot vea a un enemigo, la lista de prioridad dinámica siempre ejecutará este método. Además, el bot siempre atacará el mismo contrario, Ya que no buscará uno nuevo hasta que el objetivo actual muera o lo pierda de vista.

9.3.2. Dejar de disparar

La segunda acción en prioridad es la acción de dejar de disparar (*acción Stop_shooting*). Esta acción se ejecuta únicamente si el bot ya está disparando y no tiene ningún enemigo a la vista (*condición Must_stop_shoot*).

9.3.3. Rotar

La tercera acción consiste en girarse (*acción Rotate*). Esta acción se ejecuta (*condición Is_being_shot*) cuando el bot siente que es disparado (variable “senses”) y no tiene ningún enemigo a la vista. El bot realiza un giro de 180 grados para intentar encontrar la fuente del fuego enemigo.

9.3.4. Perseguir

La cuarta acción (*acción Pursue*) sirve para perseguir a un enemigo que se ha perdido de vista (por ejemplo ha doblado una esquina). Para que un enemigo deba ser perseguido, el bot debe tener un contrario asignado (identificado en la acción Attack), la salud por encima del límite inferior antes de comenzar a recolectar y munición en algún arma (*condición Must_pursue*). La acción consiste en el cálculo del camino hasta la última posición conocida del enemigo y la ejecución de ese camino.

9.3.5. Coger salud

La penúltima acción radica en la recolección de salud por parte del bot (*acción Get_Health*). Esta acción se ejecutará cuando la salud del bot caiga por debajo de un cierto nivel de seguridad y pueda ir a por algún objeto curativo, es decir, haya algún objeto de curación en todo el mapa que no se haya recogido (*condición Must_get_health*). Esto se puede realizar gracias a que se puede acceder a toda la información del mapa mediante el Pogamut. Para recuperar salud, el bot recogerá la información de todos los objetos curativos que haya en el mapa y caminará hacia el más cercano.

9.3.6. Coger Item

Por último, la acción menos prioritaria es la de recoger ítems (*acción Get_Items*). Esta acción se ejecutará cuando el bot perciba algún ítem en su campo de visión (*condición Can_get_items*). Al igual que en el caso anterior, el bot coge todos los ítems visibles y se encamina hacia el más cercano.

9.4. Los métodos

9.4.1. Método “prepareBot”

En este método se obtiene la instancia del BuzónGeneral que luego dirá al bot en qué equipo debe ir y se inicializan las variables correspondientes a los árboles de comportamiento: el contexto, la librería de la que se extraerá el modelo y el modelo en sí. En el contexto se introducen como variables la referencia al propio bot que lo está creando y un booleano que indica si la bandera del equipo está en su base (no hace falta distinguir el color de la bandera, eso se tratará después).

9.4.2. Método “getInitializeCommand”

Como se vio antes, este método devuelve un objeto de tipo Initialize con la información relevante del bot en este momento de su creación. En nuestro método, se accede al buzón general para obtener el equipo al que le toca pertenecer, que se introduce en el objeto Initialize. A su vez se obtienen las instancias de los buzones de ambos equipos.

9.4.3. Método “botInitialized”

Una vez que el bot está inicializado y Pogamut llama a este método, se realiza una búsqueda en todos los nodos del mapa para obtener los puntos importantes para nuestra estrategia: las bases de ambas banderas, puntos cercanos a ambas bases, puntos de patrulla, etc. Estos lugares obtenidos, sólo son válidos para el mapa con el que estamos trabajando.

Si quisiéramos introducir los bots en otro mapa, tendríamos que buscar los nodos particulares a mano para que se comportaran del mismo modo.

9.4.4. Método “botSpawned”

Una vez que el bot está a punto de aparecer en el mapa, definimos el listener para el pathExecutor para que el bot responda de manera adecuada al llegar a su destino. Por ejemplo, cuando un bot que patrulla llega al primer lugar de patrulla, se calcula la ruta hasta el segundo punto y se ejecuta el movimiento hasta ese lugar; cuando llegue al segundo punto, realizará la misma operación hasta el primero.

También se realiza la distinción del grupo al que debe pertenecer el bot dentro de su equipo, si al grupo de ataque o al de defensa y en este caso, al grupo de defensa de la bandera en la base o a la patrulla. Para ello se calcula la distancia entre el lugar en el que aparece el bot y la bandera de su equipo y entre el lugar de aparición y un punto medio del mapa de su zona; si “nace” más cerca del punto medio, le tocará ser de ataque (siempre y cuando haya sitio en el grupo de ataque), mientras que si aparece más cerca de la base, pertenecerá al grupo de defensa. La distinción entre defensa de la base o patrulla, se realiza por “pedida de turno”, es decir, el primero que llega es el primero al que le toca defender la bandera.

9.4.5. Método “logic”

En el método logic simplemente se crea el ejecutor del modelo del árbol y se le ordena ejecutar. Debido a que el logic se invoca 4 veces por segundo, es necesario controlar mediante un atributo booleano (dentro de cada nodo del árbol) si tiene que ejecutar alguna acción o no. Esto se realiza para que no se recalcule la ruta que el bot debe seguir una vez calculada por primera vez.

10.Conclusiones

El hecho de haber diseñado y desarrollado dos modelados de la IA de los agentes empleando en uno de ellos la técnica de máquina de estados y en el otro la de árboles de comportamiento para luego poder contraponerlas evaluando los pros y contras de cada una de ellas, nos ha hecho sacar nuestras propias conclusiones de ambas técnicas. Siendo las siguientes.

10.1. Árboles de comportamiento VS máquinas de estado

Por un lado están las ventajas de una máquina de estados que son: el poder descomponer acciones complejas en acciones más simples, pudiendo diseñar complejos comportamientos de una forma bastante intuitiva; la facilidad de depurar una máquina de estados gracias a la encapsulación que ofrece; y la fácil programación que conlleva.

Estas ventajas de las máquinas de estados se cumplen siempre y cuando sean máquinas pequeñas, dado que a medida que crece el número de estados para crear comportamientos más complejos surge una explosión combinatorial entre estados y transiciones bastante difícil de mantener.

Esta gran desventaja de las FSM lo solucionan en la medida de lo posible los árboles de comportamiento dado que para modelados de IAs complejas son más rápidos en ejecución sin llegar a perder la sencillez y la claridad a la hora de modelar y estructurar el diseño. A su vez, el nivel de encapsulado que éstos aportan permite una mayor facilidad en cuanto a la implementación y depurado gracias a su alta posibilidad de modularidad y abstracción en los distintos niveles de tareas.

Finalmente, gracias a todas estas ventajas que nos proporcionan los árboles de comportamiento, pudimos desarrollar mediante esta técnica y con relativa facilidad una estrategia que haga posible la cooperación entre varios agentes del mismo equipo para poder desenvolverse en una partida de captura de bandera. Hubiera resultado bastante más complejo si se hubiera diseñado con una FSM debido a la gran cantidad de situaciones que se hubieran tenido que representar mediante estados (combinaciones de cada estado de un árbol con cada estado del otro), haciendo difícil su manipulación (diseño, implementación y depuración).

Glosario

Definición de las palabras claves:

Inteligencia Artificial (IA)

Es la inteligencia dada a una máquina mediante programación con el objetivo de que se comporte de la manera más humana posible

Agente (BOT)

Personaje no controlado por un ser humano si no guiado por una IA que intenta imitar el comportamiento de un humano.

Máquina de estados (FSM)

Técnica de modelado de la IA que se compone mediante estados y transiciones. Cada estado representa una acción y cada transición una condición que hace moverse por los distintos estados.

Árboles de comportamiento (BT)

Técnica de modelado de la IA que se compone mediante tareas que pueden estar a su vez compuestas por otras tareas más simples y condiciones para saber cual de todas ellas llevar a cabo.

Unreal Tournament 2004 (UT2004)

Juego de acción en primera persona desarrollado por Epic Games.

Pogamut

Herramienta que nos permite desarrollar la IA de los bots del UT2004.


Java Behaviour Trees (JBT)

Herramienta que implementa en Java los BTs.

Bibliografía y Referencias

Mallington, Ian; Funge, John. Artificial Intelligence for Games, parte 2 capítulo 5.4. Segunda edición 2009

Palma Durán, Ricardo Juan. Java Behaviour Tree. 2010

Gemrot, J., Kadlec, R., Bida, M., Burkert, O., Pibil, R., Havlicek, J., Zemcak, L., Simlovic, J., Vansa, R., Stolba, M., Plch, T., Brom C.: Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In: Agents for Games and Simulations, LNCS 5920, [Springer](#) , 2009, pp. 1-15.

Página web wikipedia UT2004: http://en.wikipedia.org/wiki/Unreal_Tournament_2004

Página web del proyecto Pogamut: <http://diana.ms.mff.cuni.cz/main/tiki-index.php>

Wiki de Pogamut: <http://diana.ms.mff.cuni.cz/pogamut-devel/doku.php>

Repositio del JBT: <http://sourceforge.net/projects/jbt/files/>

Autorización

Los abajo firmantes: Alejandro Duchini Ordoñana y Eric Herrero Herrera, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales, y, mencionando expresamente a los autores, tanto la presente memoria, como el código, la documentación, y/o el prototipo desarrollado.

Alejandro Duchini Ordoñana

Eric Herrero Herrera

